6.0 Development Environment

The COE imposes very few requirements on the process or tools developers use to design and implement software. The COE concentrates on the end product and how it will integrate in with the overall system. This approach provides the flexibility to allow developers to conform to their internal development process requirements. However, developers are expected to use good software engineering practices and development tools to ensure robust products. The purpose of this chapter is to suggest certain development practices that will reduce integration problems, and the impact of one segment on another.

Developers may select compilers, debuggers, linkers, editors, CASE tools, etc. that are most suitable for their development environment. However, the compilers and linkers selected must be compatible with the products supplied by the hardware vendors and must not require any special products for other developers to use the segments produced.

Other specific suggestions and requirements follow:

- Segments which have public APIs written in C shall support ANSI C function prototypes.

- Segments which have public APIs shall support linking with C++ modules. This is done by bracketing function definitions with

```
#ifdef __cplusplus
extern "C" {
#endif

function prototypes

#ifdef _cplusplus
}
#endif
```

- Segments written in C that have public APIs shall handle the condition where a header file is included twice. This is accomplished by bracketing the header file with `#ifndef` and `#endif` as follows:

```
#ifndef MYHEADER
#define MYHEADER

header file declarations

#endif
```

- Code delivered to DISA shall *not* be compiled with debug options enabled, and the Unix `strip` utility shall be run on executables to minimize the disk space required.

- Segments should use shared libraries where practical to reduce runtime memory requirements. Segments with public APIs implemented as shared libraries shall also be delivered as static libraries to make debugging easier for developers who need to use the APIs.

- Developers may use GUI (Graphical User Interface) tools to build interfaces, but developer's should select tools that are portable across platforms. Segments built with such tools shall use resource files for window behavior rather than embedded code, and must not require any runtime licenses unless approved by the DISA Chief Engineer.

- Developers should run all modules through a tool such as `lint` to detect potential coding errors prior to compiling.

- Developers should run all modules through commercially available tools to detect as many runtime errors as possible (e.g., "memory leaks").

- Developers should periodically profile segments by using tools that do a runtime analysis of module performance (% CPU utilization, number of times a function is invoked, amount of time spent in a function, LAN loading analysis, etc.).

- Developers should create a test suite for automatically exercising the segment, especially inter-segment interfaces and APIs, and periodically run the tests to perform regression testing. Segments with public APIs shall be delivered with a test suite that covers all public APIs provided by the segment.

- Developers should use a tool such as `imake` for generating portable makefiles.

- Developers should use automated tools to perform configuration management tasks.

- Developers should periodically rebuild segments from scratch to ensure that all pieces, including data files, are under proper configuration management control.

6.1 Development Directory Structure

Developers may use whatever directory structure is most appropriate for their development process. The installation tools will enforce the logical structure presented in Chapter 5. However, the COE development tools allow segments under development to be located arbitrarily on the disk. For example,

```
VerifySeg -p /home5/test/dev MySeg
```

indicates that the segment to be validated, `MySeg`, is located in the directory `/home5/test/dev`. Similarly,

```
TestInstall -p /home5/test/dev MySeg
```

allows the segment to be temporarily installed from this directory for testing and debugging.

Figure 6-1 shows an example segment directory structure. It has the advantage that it separates public and private code into different subdirectories. `MySeg/libs` are public libraries provided by the segment, while `MySeg/include` are the public header files The `src/PrivLibs` subdirectory should contain library modules that are private to the segment. Similarly, the subdirectory `src/PrivInc` contains header files that are private to the segment.

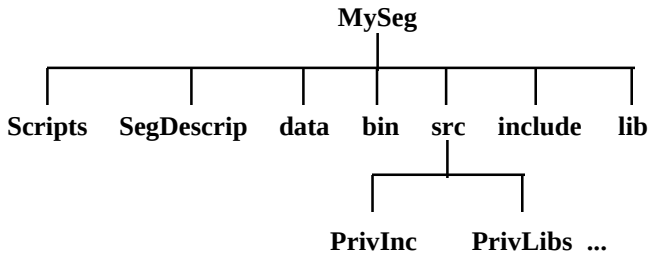

Figure 6-1: Example Development Directory Structure

The advantage of structuring directories in this fashion is that delivering software to other developers means that only one directory must be deleted: the `src` directory. Delivering the software to an operational site means that only three directories need to be deleted: `include`, `lib` (unless shared libraries are being used), and `src`. It is a simple matter to create automated scripts which can generate tapes for both types of deliveries. An additional benefit is that public

and private files are separated in the directory structure for easier management and distribution.

6.2 Development Scripts

The COE requires that a strict separation between the runtime environment and the development environment be maintained. However, it is convenient to locate development scripts in the same subdirectory as the runtime scripts (e.g., subdirectory `Scripts`). The recommended convention is to name development scripts with a `.dev` extension to distinguish them from runtime environment scripts. The `.runtime` extension can *not* be used since this has a special meaning within the COE as explained in Chapter 5.

Developers may define environment variables for locating source code directories, compilers, tools, and libraries. In addition, aliases can be defined as shortcuts for frequently executed commands. None of these examples are allowed in the runtime environment and hence must be placed in a development script such as `.cshrc.dev`.

The following suggestions are made:

- Define environment variables relative to `segprefix_HOME` where `segprefix` is the segment prefix. This allows segments to be easily relocated on the disk.

- Use environment variables to define where to place libraries and executables.

- Extend the path environment variable through concatenation - that is

  ```
  set path = ($path $TOOLS)
  ```

  where `$TOOLS` is the location of the COE development tools (e.g., `/h/TOOLS`).

- Use the same script for all supported platforms through use of the environment variables `MACHINE_CPU` and `MACHINE_OS`.

6.3 Private and Public Files

The software engineering principles of data abstraction and data hiding are important in designing segments. *Data abstraction* refers to the process of abstracting structures so that subscriber segments need not know low level details of how data is physically organized. *Data hiding* refers to hiding data elements that subscriber segments do not need, or are not authorized, to directly access. Proper implementation of these two design principles prevents segments from affecting each other through inadvertent side effects and isolates one segment from changes in another.

It is also important to hide low level functions and only provide access to segment functionality through a carefully controlled interface, the API. It is neither feasible nor desirable to make all functions in a segment available due to the sheer number of functions involved, and because changing a function that is being used directly by another developer may have significant impact.

These concepts are implemented in Ada through the *package* construct. C, however, does not contain an equivalent capability. The closest approximation in C is the *static* directive which makes a function visible only within the scope of the file containing the function definition. To compensate for structural inadequacies in C, developers must segregate software into public and private files, and into public and private directories. Since header files (e.g., `.h` files) are used to define the interface to C functions, the concept is that header files should be segregated into public and private files while public and private directories are used to provide the same concept for libraries. Moreover, segregation into distinct directories makes it easier to enforce the separation.

6.4 Developer's Toolkit

The Developer's Toolkit contains the components necessary for creating segments that use COE components. The toolkit does not need to be in segment format (it is not installed at operational sites), but is a set of files and directories that may be downloaded electronically from the on-line library. Developer's may also contact the DISA Engineering Office to receive the toolkit on magnetic media in relative "tar" format.

The Developer's Toolkit is distributed separately from the target COE-based system. However, components from the operational system (COE component segments, shared libraries, etc.) are required for development. These may be obtained electronically from the on-line library, or on magnetic media from the DISA Engineering Office. Classified or very large components will be distributed to developers via magnetic media. The toolkit does not duplicate any components available in the runtime system because this would create configuration management problems in ensuring that developers do not receive two different versions of the same module.

As distributed, the toolkit contains the following:

- API libraries and object code
- C header files for public APIs
- On-line Unix `man` pages for APIs
- COE development tools (see Appendix C)
- Standards for creating APIs

The toolkit does *not* contain any products which require a license (compilers, editors, RDBMS, etc.). It is the developer's responsibility to acquire these items as needed.

Developers may install the toolkit on the disk in whatever directories are desired. The standard location for toolkit components are:

public header files	`/h/COE/include`
public libraries	`/h/COE/lib`
executables	`/h/TOOLS/bin`
man pages	`/h/TOOLS/man`

Certain tools from Appendix C are useful for both the development environment and the runtime environment. These tools are delivered with the operational system and are located under `/h/COE/bin`.

Developers should include /h/TOOLS/bin in the path environment variable for their development environment. /h/TOOLS/man should also be included in the search path for Unix man pages.

Developers are encouraged to submit tools to the COE community for inclusion in the developer's toolkit. All tools submitted must be license and royalty free, and must include a man page for on-line documentation. Developers wishing to release source code for their contributed tools may do so, and the source code for the tool will be organized under the /h/TOOLS/src directory.